JL

LEVEL II (1)

(6) DETERMINISTIC MODELS

(10) Chung Laung /Liu see AD A110229

Departement of Computer Science
University of Illinois
Urbana, Illinois, USA

(11) 1981

(12) 21

There are several reasons for studying deterministic models of computer systems and their work loads[†]:

(1) With deterministic models, we can carry out a worst-case or a best-case analysis so that we can obtain upperbounds or lower-bounds on the performance of a system under all possible circumstances;

(2) Effects of variation of system parameters can be studied more directly and explicitly;

(3) There exists the possibility of designing optimal algorithms for the effective utilization of system resources. Such algorithms often behave well when the systems deviate from the (deterministic) model.

We present here three closely related topics to illustrate some of the aspects of deterministic modelling.

[†] This work was partially supported by the US Office of Naval Research under the Contract Number N00014-79-C-0775

(15)

176 011

81 7 30 059

I - A MODEL

We describe first a general model of computing system which can be specialized in various ways to include most of the results we are going to present. We make the following assumptions :

(1) A computing system consists of two classes of resources, dedicated resources and shared resources. In each class, there are different kinds of resources.

(2) There is a certain number of units of dedicated resources of each kind. The execution of a job requires an integral number of units of each kind, including zero unit as a possibility. The execution of a job completely occupies a unit, and no other jobs can be executed on the same unit concurrently. Examples of dedicated resources are processors, input-output devices, and so on.

(3) There is a unit of shared resources of each kind[†]. The executon of a job requires a fraction ot the unit of each kind of shared resources, including zero as a possibility. Concurrent execution of a number of jobs might share the same unit of shared resources, provided that the sum of the fractions of the unit they share does not exceed one. Examples of shared resources are core memories, magnetic disks and drums, and so on.

(4) The units of each kind of dedicated resources might not be identical. It might be the case that the execution times will be different when a job is executed on different units of one kind of dedicated resources. It might be the case that job can only be executed on some of the units of a particular kind of dedicated resource. Since the execution of a job might require, in general, more than one unit of dedicated resources of each kind, the execution of a job is said to be completed if its execution on all units is completed.

(5) The unit of each kind of shared resources is considered to be uniform. A job will release the portions of shared resources it occupies when its execution on all units of dedicated resources is completed.

[†] There is no loss in generality in normalizing each kind of shared resources to one unit.

II - A MODEL

We describe first a general model of computing system which can be specialized in various ways to include most of the results we are going to present. We make the following assumptions :

(1) A computing system consists of two classes of resources, dedicated resources and shared resources. In each class, there are different kinds of resources.

(2) There is a certain number of units of dedicated resources of each kind. The execution of a job requires an integral number of units of each kind, including zero unit as a possibility. The execution of a job completely occupies a unit, and no other jobs can be executed on the same unit concurrently. Examples of dedicated resources are processors, input-output devices, and so on.

(3) There is a unit of shared resources of each kind[†]. The executon of a job requires a fraction ot the unit of each kind of shared resources, including zero as a possibility. Concurrent execution of a number of jobs might share the same unit of shared resources, provided that the sum of the fractions of the unit they share does not exceed one. Examples of shared resources are core memories, magnetic disks and drums, and so on.

(4) The units of each kind of dedicated resources might not be identical. It might be the case that the execution times will be different when a job is executed on different units of one kind of dedicated resources. It might be the case that job can only be executed on some of the units of a particular kind of dedicated resource. Since the execution of a job might require, in general, more than one unit of dedicated resources of each kind, the execution of a job is said to be completed if its execution on all units is completed.

(5) The unit of each kind of shared resources is considered to be uniform. A job will release the portions of shared resources it occupies when its execution on all units of dedicated resources is completed.

[†] There is no loss in generality in normalizing each kind of shared resources to one unit.

Let $J = \{J_1, J_2, \ldots, J_i, \ldots\}$ be a set of jobs and $<$ be a precedence relation on J[+]. That $J_K < J_\ell$ means the execution of job J_ℓ cannot begin until the execution of job J_K has been completed. J_K is called a predecessor of J_ℓ, and J_ℓ is called a successor of J_K. A set of job is said to be independent if the precedence relation $<$ is empty. Each job in J is specified in the following way : Let p denote the number of kinds of dedicated resources and n denote the number of units of each kind[*] in the computing system on which the set of jobs J is to be executed. Let q denote the number of kinds of shared resources. The utilization of the dedicated resources by a job J_K is specified by a $p \times n$ matrix $T_K = \|t_{ij}\|$ and a p component vector $U_K = \|u_i\|$, where $0 \leq t_{ij} \leq \infty$ and for each i there exists at least one j such that $t_{ij} < \infty$, and u_i is an integer such that $0 \leq u_i \leq n$. The value of t_{ij} is the time it takes to execute job J_K on the jth unit of dedicated resources of the ith kind. That $t_{ij} = \infty$ means that job J_K cannot be executed on the jth units of dedicated resources on the ith kind. The value of u_i is the number of units of dedicated resources on the ith kind which the execution of job J_K requires.[**] Similarly, the utilization of the shared resources by J_K is specified by a q component vector $V_K = \|v_i\|$, where $0 \leq v_i \leq 1$. The value of v_i is the fraction on the ith kind of shared resources which the execution of job J_K requires.

II - SCHEDULING TO MINIMIZE COMPLETION TIME

By scheduling a set of jobs on a computing system, we mean to assign, within certain time interval(s), to each job resources that are needed for its execution with the constraint that all the resources needed for the execution of a job are assigned to the job simultaneously. A schedule is a specification of the assignment of resources to the jobs, and a scheduling algorithm is a procedure that produces a schedule for every given set of jobs. By preemptive scheduling discipline, we mean to allow the interruption of the execution of jobs in a schedule. By non-preemptive scheduling discipline, we mean the execution of a job must continue until completion, once its execution commences.

Different criteria can be used to measure how good a schedule is. The most common one is the completion time of a schedule, that is, the total time it

[+] A precedence relation is a binary relation that is antisymmetric and transitive.

[*] As will be seen, there is no loss of generality in assuming that there is the same number of units in each kind of dedicated resources.

[**] For each i, u_i is not larger than the number of finite entries in the ith row of T_K.

takes to complete the execution of a set of jobs according to the schedule. Clearly, for a given set of jobs, a "good" schedule is one with "short" completion time, and an <u>optimal</u> schedule is one with shortest possible execution time. The effectiveness of a scheduling algorithm is measured by how good the schedules it produces are. One might wish to consider the worst case performance of a scheduling algorithm, or one might wish to consider the average case performance of a scheduling algorithm. Most of the current works are concerned with the worst case performance analysis of scheduling algorithms. We shall make an attempt to identify some of the general features of scheduling algorithms whose effectiveness will be measured by the completion time of the schedules they produce.

1. <u>There are algorithms that produce optimal schedules</u>. Clearly, optimal schedules and algorithms that produce optimal schedules are of significant interest. Unfortunately, very little is known about "efficient" algorithms that produce optimal schedules for arbitrary computing systems and arbitrary sets of jobs. As a matter of fact, efficient algorithms that produce optimal schedules are known only for the following cases :

(i) Jobs having unit execution times with the precedence relation over them being a forest are to be scheduled on a computing system with identical processors.

(ii) Jobs having unit execution times are to be scheduled on a computing system with two identical processors.

We shall describe an algorithm due to Hu [H3] which produces an optimal schedule for case (i). We introduce first the notion of <u>demand scheduling algorithm</u>. A demand scheduling algorithm is one that always attemps to schedule executable jobs[+] on resources that are free at any time instant. In other words, a demand scheduling algorithm never leaves any resources idle intentionally.[‡] A particularly simple class of demand scheduling algorithms is known as <u>list scheduling algorithms</u>. A list scheduling algorithm assigns distinct priorities

[†]A job is said to be executable at a time instant if executions of its predecessors have all been completed at that time instant.

[‡]It is not difficult to construct examples to show that there are optimal schedules in which resources are left idle intentionally.

to jobs and allocates resources to jobs with highest priorities among all executable ones at any time instant.

Hu's algorithm is a list scheduling algorithm. We define first the notion of the <u>level</u> of a job :

(i) The level of a job that has no successor is defined to be 1.

(ii) The level of a job that one or more successors is equal to one plus the maximum value of the levels of its successors.

In Hu's algorithm priorities are assigned to jobs according to their levels such that jobs of higher levels will have higher priorities. (Assignment of priorities to jobs of the same level is arbitrary.) Hsu [H2] contains a simple proof that Hu's algorithm produces optimal schedules for case (i). See also Chen and Liu [4].

Fujii, Kasami, and Ninomiya [F2] and Coffman and Graham [C7] discovered algorithms that produce optimal schedules for case (ii). We present here Coffman and Graham's algorithm, which is also a list scheduling algorithm. In Coffman and Graham's algorithm, priorities are assigned to jobs as follows :

(i) Starting with 1, which is the lowest priority, distinct and consecutive priorities are assigned to jobs that have no successors arbitrarily.

(ii) Priorities are assigned to jobs with one or more successors recursively :

(a) A job to all of whose successors priorities have been assigned will be labelled with the priorities of its successors $(i_1, i_2, \ldots)$ in decreasing order.

(b) Compare the labels of all labelled jobs according to the lexicographical order. Starting with the lowest unassigned priority, distinct and consecutive priorities are assigned to the labelled jobs such that jobs with larger labels will be assigned higher priorities.

We should point out that there is a large body of literatures on obtaining optimal schedules by the methods of complete enumeration, mixed integer and nonlinear programming, and dynamic programming. Note that in these approaches, the computation time required to produce an optimal schedule will be an exponential function of the number of jobs to be scheduled. We refer the reader to Lenstra [L4] and Rinooy Kan [R1]. See also Horowitz and Sahni [H1] and Sahni [S1].

2. <u>There are simple scheduling algorithms that spend very little effort to search for a schedule</u>. Almost directly opposite to the approach of spending a lot of effort to determine an optimal schedule, one could consider the approach of spending little or no effort to search for a reasonably good schedule. In view of the discovery of the class of NP-complete problems, such an approach becomes a particularly attractive one. (As general references to the area of approximation algorithms, see Johnson [J1] and Garey and Johnson [G2]). For example, a very simple scheduling algorithm is a list scheduling algorithm with arbitrary assignment of priorities. The following result is due to Graham [G6, G7, G8] :

<u>Theorem 1</u> : For a computing system with n identical processors, let ω denote the completion time of a schedule for a given set of jobs produced by an arbitrary list scheduling algorithm and let ω_0 denote the shortest possible completion time.[+] Then

$$\frac{\omega}{\omega_0} \leq 2 - \frac{1}{n}$$

For $n = 2$, the ration ω/ω_0 in Theorem 1 is upperbounded by the constant 3/2. That is, in terms of the completion time a schedule produced by any list scheduling algorithm is not worse than an optimal schedule by 50%. When the number of processors in the system increases, although the comparison becomes less favorable, the sub-optimal schedule is never worse than an optimal schedule by 100%.

[+]From now on, we shall consistently use ω to denote the completion time of an arbitrary schedule and ω_0 to denote the completion time of an optimal schedule.

Theorem 1 can be extended immediately :

Theorem 2 (Liu and Liu [L7]):For a computing system with n_1 processors of speed b_1, n_2 processors of speed b_2, ..., n_k processors of speed b_k, where $b_1 > b_2 > \dots > b_k \geqq 1$, we have

$$\frac{\omega}{\omega_0} \leqq \frac{b_1}{b_k} + 1 - \frac{b_1}{\sum_{i=1}^{k} n_i b_i}$$

Garey and Graham [G1], and Yao [Y1] studied list scheduling algorithms for computing systems with shared resources. For example, similar to Theorems 1 and 2, we have.

Theorem 3 (Garey and Graham [G1]) : For a computing system with n identical processors and one kind of shared resources, we have

$$\frac{\omega}{\omega_0} \leqq n$$

Theorem 4 (Garey and Graham [G1]) : For a computing system with two or more processors and q kinds of shared resources and for a set of independent jobs, we have

$$\frac{\omega}{\omega_0} \leqq \min\left(\frac{n+1}{2},\ q + 2 - \frac{2q-1}{n}\right)$$

See also [I1, K1, L7, L8, L9] where various extensions of the case where the processor are not identical were studied.

3. There are cases in which an algorithm that produces optimal schedules under a certain set of conditions is applied to situations that do not satisfy these conditions. The following results illustrate this point :

Theorem 5 (Chen [C2]) : When Hu's algorithm is applied to schedule a set of jobs with unit execution times on a computing system with n identical processors. Then

$$\frac{\omega}{\omega_0} \leqq \frac{4}{3} \qquad n = 2$$

$$\frac{\omega}{\omega_0} \leqq 2 - \frac{1}{n-1} \qquad n \geqq 3$$

Theorem 6 (Lam and Sethi [42]) : When Coffman and Graham's algorithm is applied to schedule a set of jobs with unit execution times on a computing system with n identical processors. Then

$$\frac{\omega}{\omega_0} \leq 2 - \frac{2}{n}$$

Kaufman [K5] extended Hu's algorithm to the scheduling of jobs with unequal execution times on a computing system with n identical processors, where the precedence relation over the jobs is a forest. By defining the level of a job to be the length of the chain between the job and the root of the tree it is in (including the execution time of the job itself), Kaufman has shown that

Theorem 7 : In the extended Hu's algorithm described above

$$\omega \leqq \omega_p + k - k/n$$

where ω_p is the completion time when the jobs are executed according to an optimal preemptive schedule, and k is the execution time of the longest job in the set.

4. There are algorithms that perform a certain amount of computation in order to produce good schedules. For example, consider the problem of scheduling a set of independent jobs on a computing system with n identical processors. If we sort the jobs according to their execution times and assign high priorities to jobs with long execution times, we can upperbound the worse case behavior of such a list scheduling algorithm by :

Theorem 8 (Graham [G7]) : For the scheduling algorithm described above

$$\frac{\omega}{\omega_0} \leqq \frac{4}{3} - \frac{1}{3n}$$

Extension of the idea of assigning high priorities to jobs with long execution times to computing systems with non-identical processors have been carried out in Gonzales, Ibarra, and Sahni [G9], and Ibarra and Kim [I1] .

As another example, we consider the following algortihm for scheduling a set of independent jobs on a computing system with n identical processors : We pick out the k longest jobs in the set and schedule them in such a way that the total execution time (for the execution of these k jobs) is minimum. The remaining jobs will be scheduled according to the rule that whenever a processor is free an arbitrarily chosen job will be executed on that processor. Graham [G7] has shown that

<u>Theorem 9</u> : For the scheduling algorithm described above

$$\frac{\omega}{\omega_0} \leq 1 + \frac{1 - \frac{1}{n}}{1 + \left\lfloor \frac{k}{n} \right\rfloor}$$

5. <u>One can consider algorithms that produce schedules which are as close to optimal schedules as it is desired at the expense of computation time.</u> An algorithm is said to be an <u>ε-approximation algorithm</u> if for a given ε the algorithm will yield a schedule such that the ratio $(\omega-\omega_0)/\omega_0$ is less than ε. Sahni [S1] studied the problem of scheduling a set of independent jobs on a computing system with n identical processors and obtained an ε-approximation algorithm whose complexity is $\mathcal{O}(m(m^2/\epsilon)^{n-1})$ where m is the number of jobs is the set.

III - <u>SCHEDULE TO MEET DEADLINES</u>

To illustrate some other aspects of the scheduling problem, we shall survey some of the results on a generalization of our model by assuming that each job has a <u>ready time</u>, a time at or after which execution of the job can begin, and a <u>deadline</u>, a time at or prior to which execution of the job must be completed.For a given computing system with fixed amounts of resources, a set of job is said to be <u>schedulable</u> if there is a schedule according to which all jobs can be executed to meet their deadlines. Such a schedule will be referred to as a <u>feasible schedule</u> for the set of jobs. A set of jobs is said to be <u>schedulable</u> by a scheduling algorithm if the algorithm yields a feasible schedule for the set. Consequently, a scheduling algotithm is said to be <u>optimal</u> if it yields a feasible schedule for every schedulable set of jobs. On the other hand, if a scheduling algorithm is not optimal, one would wish to measure the effectiveness of the algorithm is terms of the fraction of schedulable sets of jobs it is capable of scheduling.

1. Optimal scheduling. Garey and Johnson [G4, G5] obtained an optimal scheduling algorithm for the following case :

(i) The computer system has two identical processors.

(ii) Each job has unit execution time.

(iii) Each job has a prespecified ready time and deadline.

(iv) There is an arbitrary precedence constraint over the jobs.

Garey and Johnson's algorithm yields a schedule that enables the completion of each job before each deadline if such a schedule exists.

2. Real-time scheduling. In many cases when a computer is used for control or monitoring functions, the following model is encountered.

(i) There is a single processor in the computer system.

(ii) Each job has a ready time and a deadline.

(iii) The execution of a job can be preempted by a another job.

In [L1, L6, S3], an algorithm know as earliest deadline first algorithm has been shown to be optimal. The earliest deadline first algorithm always execute a job that has the earliest deadline among all the ready jobs. Thus, a newly arrived job will preempt a job that is currently being executed if the new arrival has an earlier deadline. The earliest deadline first algorithm is optimal in the sense that if a set of jobs can be scheduled by any algorithm, it can also be scheduled by the earliest deadline first algorithm.

Some variations of the pr blem are :

(i) the execution of a job cannot be preempted.

(ii) There are two or more processors in the computer system.

(iii) There is a precedence constraint over the jobs.

(iv) The execution of a job requires other kinds of resources.

3. Jobs with Periodic Requests

A special case of real time scheduling was studied in [L6]. In this case, a job consists of a periodic stream of requests. That is, a job J_i demands periodically C_i units of computation time in every T_i units of time (T_i is referred to as the request period), and the deadline of a request is assumed to be the ready time of the next request of the same job. Although the earliest deadline first scheduling algorithm can be applied to this case, another scheduling algorithm known as the rate monotonic algorithm has been studied. In the rate monotonic algorithm, requests of a job with the shortest request period always have priority over the requests of a job with a longer request period. Not only such an algorithm is easy to implement, its simplicity also enables us to carry out a more thorough analysis of its performance. We have

Theorem 1 : Any set of n jobs with $\sum_{i=1}^{n} \frac{C_i}{T_i} < n\,(2^{1/n} - 1)$ can be scheduled by the rate monotonic algorithm to meet all deadlines. Furthermore, there is a set of n jobs with $\sum_{i=1}^{n} \frac{C_i}{T_i} > n\,(2^{1/n} - 1)$ that cannot be scheduled by the rate monotonic algorithm.

Theorem 1 is again another example illustrating the possibility of lower bounding the performance of a system. Note that C_i/T_i is the percentage of time the jobs J_i will utilize the processor. Consequently, Theorem 1 says that if a set of jobs doesnot try to utilize the processor beyond a certain percentage, the rate monotonic algorithm can guarantee that all deadlines will be met . (Specifically, for n=2, $n\,(2^{1/n} - 1) = 0.828$, for $n \to \infty$, $n\,(2^{1/n} - 1) \to 0.63$). Note that there are sets of jobs with a total utilization above the bound which can still be scheduled by the rate monotonic algorithm. However, our bound provides a guarantee under the worst possible situation.

Another resuet of the same flavor is an estimation on the slack time of a request which is defined to be the time span between the completion of the execution of a request and its deadline. In many practical situation, not only do we wish to meet all the deadlines, we also would like to have a large slack time if possible. We can show that [L5] :

Theorem 2 : If the first requests of all jobs occur at $t = 0$, then the slack time of any request is larger than or equal to the slack time of the first request of the same job.

Theorem 3 : For a set of n jobs with $\sum_{i=1}^{n} \frac{C_i}{T_i} < n\,(2^{1/n} - 1)$, the slack time of any request is larger than a equal to 0.207 g where g is the last quantum of processor time allocated to the first request of the same job.

Theorem 4 : For a set of n jobs wich $\sum_{i=1}^{n} \frac{C_i}{T_i} < n\,(2^{1/n} - 1)$ and with $T_n \geq 2\,T_{n-1}$, the slack time of any request of J_n is larger than or equal to 0.207 g where g is the last quantum of processor time allocated to that request.

Again, note the possibility of lower bounding the performance of a system under some very general conditions.

IV - BIN PACKING ALGORITHM

The bin packing problem can be described as placing a list of "pieces" of size larger than 0 and less than or equal to 1 into "bins" of size 1 so as to minimize the total number of bins utilized. There are some immediate interpretations of the bin packing problem :

(i) Table formatting : To place items of data (pieces) in computer words of fixed size (bins).

(ii) Prepaging : To place program segments (pieces) into pages (bins).

(iii) File allocation : To place files (pieces) on disk tracks (bins).

A bin packing algorithm is said to be optimal if it uses the minimum number of bins. On the other hand, the performance of a suboptimal algorithm can be measured by the quantity $\lim_{k\to\infty} F(k)$, where for a fixed k F(k) is the maximum number of bins used by the algorithm over all possible lists of pieces that can be packed into k bins by an optimal algorithm divided by k.

1. On-line Algorithms

A bin packing algorithm is said to be an on-line algorithm if the pieces are available one at a time and a piece must be assigned to a bin before the next one becomes available. We mention first three well-known algorithms :

Next fit (NF) algorithm : A piece will be place into the "current bin" if it can be fit into that bin. If not, a new bin will be used and will be designated the current bin.

First-fit (FF) algorithm : the bins will be indexed B_1, B_2,... A piece will be placed into a bin B_j that can accomodate it with the smallest index j.

Best-fit (BF) algorithm : The bins will be indexed B_1, B_2... A piece will be placed into a bin that has been filled to a highest possible level and can still accomadate the piece. If there are more than one such bin, choose the one with the smallest index j.

We have [J3] :

Theorem 1 :

$$\lim_{k\to\infty} NF(k) = 2$$

$$\lim_{k\to\infty} FF(k) = \frac{17}{10}$$

$$\lim_{k\to\infty} BF(k) = \frac{17}{10}$$

A new on-line algorithm was proposed by Yao [Y2] which is known as the refined first fit (RFF) algorithm. The refined first fit algorithm can be described as follows :

(1) A piece will be called an A-piece, B_1-piece, B_2-piece, or X-piece if the size of the piece is in the interval $(1/2, 1]$, $(2/5, 1/2]$, $(1/3, 2/5]$, or $(0, 1/3]$, respectively.

(2) The set of all bins are divided into four infinite classes, to be referred to as class 1, 2, 3, 4, respectively.

(3) Let m be a fixed integer whose value can be chosen as 6, 7, 8, or 9.

(4) Suppose the first j-1 pieces have been assigned. The j^{th} piece will be assigned according to the first-fit algorithm into a bin of a certain class. In particular,

(i) If it is an A-piece, it will be assigned to a class 1 bin.

(ii) If it is a B_1-piece, it will be assigned to a class 2 bin.

(iii) If it is a B_2-piece, but <u>not</u> the (mi)-th B_2-piece seen so far for some integer $i \geq 1$, it will be assigned to a class 3 bin. If it is the (mi)-th B_2-piece for some integer $i \geq 1$, it will be assigned to a class 1 bin containing an A-piece if possible, or to a new class 1 bin otherwise.

It can be shown that [Y2] :

<u>Theorem 2</u> : $\lim_{k \to \infty} RFF(k) = \frac{5}{3}$

Yao [Y2] has also obtained a lower bound on the performance of any on-line bin packing algorithm :

<u>Theorem 3</u> : For any on-line packing algorithm S, $\lim_{k \to \infty} S(k) \geq \frac{3}{2}$.

2. <u>Off-line Algorithms</u>

A bin packing algorithm is said to be an off-line algorithm if all the pieces for packing are available before commencing.

First-fit decreasing (FFD) algorithm : the pieces are arranged is non-incresing order according to size and then apply the first-fit algorithm.

Best-fit decreasing (BFD) algorithm : The pieces are arranged in non-increasing order according to size and then apply the best-fit algorithm.

Theorem 4 [J3] :

$$\lim_{k \to \infty} FFD(k) = \frac{11}{9}$$

$$\lim_{k \to \infty} BFD(k) = \frac{11}{9}$$

3. Variation of Parameters

For a give system with a certain performance index, one often would want to know how the performance index varies as one or more of the parameters of the system vary. Such information will be useful, for example, in determining the cost-effectiveness of tuning a system by varying some of the parameters. A study by Friesen [F1] on bin packing algorithms provides an interesting example. For a given bin packing algorithm, one might ask how the algorithm behaves if instead of packing the pieces into bins of size 1 (the standard size) we shall pack the pieces into bins of size α, $\alpha > 1$. This is exactly the case of determining the performance of a paging algorithm when the size of a page is increased. For example, an interesting question is how big the size of a page should become so that a suboptimal paging algorithm will not used more pages than an optimal paging algorithm when the latter uses pages of standard size. For a fixed k, let $FFD_\alpha(k)$ denote the maximum number of bins of size α used by the first fit decreasing algorithm over all possible lists of pieces that can be packed by an optimal algorithm using k bins of standard size (1) divided by k. We have [F1] :

Theorem 5

α	$\lim_{k \to \infty} FFD_{\alpha}(k)$
[1, 45/44)	11/9
[45/44, 30/29)	29/24
[30/29, 25/24)	6/5
[25/24, 20/19)	19/16
[20/19, 8/7)	7/6
[8/7, 15/13)	13/12
[15/13, 36/31)	31/30
[36/31, 48/41)	41/40
[48/41, 72/61)	61/60
[72/61, 2)	1

REFERENCES

[B1] Baer, J. L., "A survey of some theoretical aspects of multiprocessing," Computing Surveys, 5 (1973), 31-80.

[B2] Baker, K., Introduction to Sequencing and Scheduling, John Wiley & Sons, 1974.

[B3] Błazewicz, J., "Deadline scheduling of tasks with ready times and resource constraints," Info. Processing Letters, 8 (1979), 60-63.

[B4] Błazewicz, J, and J. Weglarz, "Scheduling under resource Constraints - achievements and prospects," Performance of Computer Systems, Conference Preprints, 4th International Symposition on Modelling and Performance Evaluation of Computer systems Vol. 2,1979.

[B5] Brucker, P., J. K. Lenstra and A. H. G. Rinnooy Kan, "Complexity of machine scheduling problems," Oper. Res.

[C1] Chandra, A. K., and C. K. Wong, "Worst-case analysis of a placement algorithm related to storage allocation," SIAM J. Computing, 4 (1975), 244-263.

[C2] Chen, N. F., "An analysis of scheduling algortithms in multiprocessing computing systems," Technical Report VIVCDCS-R-75-724, Department of Computer Science University of Illinois at Urbama-Champaign, 1975.

[C3] Chen, N. F. and C. L. Liu, "On a class of scheduling algorithms for multiprocessors computing systems," Proceedings of the 1974 Sagamore Computer Conference on Parallel Processing (1974), 1-16.

[C4] Coffman, E. G., Jr., (ed.), Computer and Yob-Shop scheduling Theory, John Wiley & Sons, 1976.

[C5] Coffman, E. G., Jr. and P. J. Denning, Operating Systems Theory, Prentice-hell, 1973.

[C6] Coffman, E. G., Jr., M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," SIAM J. on Computing, 7 (1978), 1-17.

[C7]Coffman, E. G., Jr. and R. L. Graham, "Optimal scheduling for two processor systems," Acta Informatica, 1 (1972), 200-213.

[C8]Conway, R. W. L. Maxwell and L. M. Miller, Theory of Scheduling, Addison-Wesley, 1967.

[D1]Dhall, S. K., and C. L. Liu, "On a real-time scheduling problem," Operation research, 26 (1978), 127-140.

[F1]Friesen, D. K., "Sensitivity Analysis for heuristic algorithms," Technical Refort, UIUCDCS-R-78-934, August 1978, Department of Computer Science, University of Illinois at Urbana-Champaign.

[F2]Fujii, M., T. Kasami, and K. Ninomiya, "Optimal sequence of two equivalent processors," SIAM J. on Applied Math., 17 (1969), 784-789. Erratum, 20 (1971), 141.

[G1]Garey, M. R., and R. L. Graham, "Bounds for multiprocessor scheduling with resource constraints, "SIAM J. on Computing, 4 (1975), 187-200.

[G2]Garey, M. R., R. L. Graham, and D. S. Johnson, "Performance guarantees for scheduling algorithms," Operations Research, 26 (1978), 3-21.

[G3]Garey, M. R., R. L. Graham, D. S. Johnson, and A. C. Yoa, "Multiprocessor scheduling as generalized bin-packing," J. Comb. Th. 21 (1976), 257-298.

[G4]Garey, M. R., and D. S. Johnson, "Scheduling tasks rith monuniform deadlines or two processors," J. ACM 23 (1976), 461-467.

[G5]Garey, M. R., and D. S. Johnson, "Two-processors scheduling with start-times and deadlines," SIAM J. on Computing, 6 (1977), 416-426.

[G6] Graham, R. L., "Bounds for certain multiprocessing anomalies," Bell Sys. Tech. J., 45 (1966), 1563-1581.

[G7] Graham, R. L., "Bounds on multiprocessing anomalies," SIAM J. on Applied Math., 17 (1969), 416-429.

[G8] Graham, R. L., "Bounds on multiprocessing anomalies and related packing problems," Proc. of the Spring Joint Computer Conference (1972), 205-217.

[G9] Gonzales, T., O. H. Ibarra, and S. Sahni, "Bounds for LPT schedules on uniform processors," SIAM J. on Computing, 6 (1977), 155-166.

[G10] Gonzales, T., and S. Sahni, "Preemptive scheduling of uniform processor systems," J. ACM, 25 (1978), 92-101.

[H1] Horowitz, E., and S. Sahni, "Exact and approximate algorithms for scheduling non identical processors," J. ACM, 23 (1976), 317-327.

[H2] Hsu, N. C., "Elementary proof of Hu's theorem on isotine mappings," Proc. AMS, 17 (1966), 111-114.

[H3] Hu, T. C., "Parallel scheduling and assembly line problems," Oper, Res., 9 (1961), 841-848.

[I1] Ibarra, O. H., and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on non-identical processors," J. ACM, 24 (1977), 280-289.

[J1] Johnson, D. S., "Approximation algorithms for combinstorial problems," J. Computer and Systems Sciences, 9 (1974), 256-278.

[J2] Johnson, D. S., "Fast algorithms for bin packing," J. Computer and Systems Sciences, 8 (1974), 272-314.

[J3] Johnson, D. S., A. Demars, J. D. Ullman, M. R. Garey, and R. L. Graham, Worst-case performance bounds for simple one dimentional packing algorithms," SIAM J. on Computing, 3 (1974), 299-325.

[K1] Kafura, D. G., "Analysis of scheduling algorithms for a model of a multiprocessing computer system," Ph. D. Thesis, Purdue University, 1974.

[K2] Kafura, D. G., and V. Y. Shen, "Scheduling independent processors with different storage capacities," Proc. ACM National Conf. (1974), 161-166.

[K3] Kafura, D. G. and V. Y. Shen, "An algorithm to design the memory configuration of a computer network," Purdue University, Computer Science Technical Report, 1975.

[K4] Kafura, D. G., and V. Y. Shen, "Task scheduling on a multiprocessor system with independent memories," SIAM J. on Computing, 6 (1977), 167-187.

[K5] Kaufman, K. T., "An almost-optimal algorithm for the assembly line scheduling problem," IEEE Trans. on Comp., C-23 (1974), 1169-1174.

[K6] Krause, K. L., V. Y. Shen, and H. D. Schwetman, "Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems" J. ACM, 22 (1975), 522-550.

[L1] Labetoulle, J., "Some theorems on real time scheduling, "Computer Architectures and Networks, E. Gelenbe and R. Mehl ed., North-Holland Publ. Co., 1974, 285-298.

[L2] Lam, S., and R. Sethi, "Worst case analysis of two scheduling algorithms," SIAM J. on Computing, 6 (1977), 518-536.

[L3] Lawler, E. L., and J. Labetoulle, "On preemptive scheduling of unrelated parallel processors by linear programming," J. ACM, 25 (1978), 612-619.

[L4] Lenstra, J. K., Sequencing by Enumerative Methods, Mathematisch Centrum, Amsterdam, 1976.

[L5] Liu, C. L., J. W. S. Liu, and A. Liestman, "Scheduling with slack time," (to appear).

[L6] Liu, C. L., and J. W. Layland, "Scheduling algorithms for multiprogramming in a Hard-real-time environment," J. ACM, 20 (1973), 46-61.

[L7] Liu, J. W. S., and C. L. Liu, "Bounds on scheduling algorithms for heterogeneous computing systems," Proc. of the 1974 IFIP Cong. (1974), 349-353.

[L8] Liu, J. W. S., and C. L. Liu, "Performance analysis of multiprocessor systems containing functionally dedicated processors," Acta Informatica, 10 (1978), 95-104.

[L9] Liu, J. W. S., and A. Yang, "Optimal scheduling of independent tasks on heterogeneous computing systems," Proc. of the ACM (1974), 38-45.

[R1] Rinooy Kan, A. H. G., Machine Scheduling Problems, H. E. Stenfert Kroese B. V., Leiden, 1974.

[S1] Sahni, S., "Algorithms for scheduling independent tasks," J. ACM, 23 (1976), 116-127.

[S2] Schindler, S., "On optimal schedules for multiprocessor systems," Proc. of Princeton Conf. on Information, Scienes and Systems (1972), 219-223.

[S3] Serlin, O., "Scheduling of time critical processes," Proc. of the Spring Joint Computers Conference (1972), 925-932.

[Y1] Yao, A. C., "On scheduling unit-time tasks with limited resources," Proc. of the Sagamore Conference on Parallel Processing (1974), 17-36.

[Y2] Yao, A. C., "New algorithms in bin packing," Technical Report STAN-CS-78-662, September 1978, Computer Science Department, Stanford University.